CIS 330
C/C++ and Unix

Lecture 10

Initialization and Cleanup

Tech Together

Saturday, May 7th
10am - 1:30pm
EMU | Crater Lake Room

Brunch, panel of UO Alumni working in tech

Use this QR code to register and learn more:

We hope to see you there!

Tech Together

Raffle!

- $50 Duckstore giftcards
- Starbucks giftcards
- Exclusive career coaching session with career coach, Wendy Bohling

Register at wicsuo.wixsite.com/uowics or go to @uowics on Instagram and click the link in our bio!

Review

- Private member cannot be accessed through objects
- Constructors are destructors are used to initialize/cleanup objects
- Inheritance and polymorphism
- Friendship can be used to give access to non-members of a class
- "Cheshire cat smile" can be used to limit access to implementation details

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Header File

```
1. #ifndef HANDLE_H
2. #define HANDLE_H
3. class Handle {
4.     struct Cheshire;
5.     Cheshire *smile;
6. public:
7.     void initialize();
8.     void cleanup();
9.     int read();
0.     void change(int);
1. };
2. #endif
```

Header file can be seen by anyone – otherwise you won't know how to use it

The REAL private members are not visible

The real private members are "hidden" inside the Cheshire struct

Cheshire Cat

- Implement the Class in the source code
- Compile it
- Provide it as a library/API with the header to provide the interface (and include what the interface is used for)
- The user has no idea what's actually inside the Class (i.e., the private members)

Cheshire Cat

```
#include "Handle.h"
struct Handle::Cheshire {
  int i;
};
void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}
void Handle::cleanup() {
  delete smile;
}
int Handle::read() {
  return smile->i;
}
void Handle::change(int x) {
  smile->i = x;
}
```

Questions?

Initialization and Cleanup

Lot of bugs can happen due to initialization (or lack thereof)

- In the SpMV homework, some people were getting errors because they forgot to initialize the output vector to 0
- += to a random number results in a random number

More cleanup is required (beyond `free()`) in C++ due to the complexity of structure/classes

Constructors and Destructors guarantee that data is initialized and destroyed appropriately

`this` Pointer

To talk about initialization, it's important to talk about memory

- Each object gets its own copy of data members but share a single copy of the member functions (modularity)

Then, how are data members accessed updated using single copy of the functions?

- The '`this`' pointer is passed (as a hidden argument) to all non-static member function calls, and is available as a local variable within the function body
- For example, for class `X`, '`this`' pointer is of type X*

Recall that `static` members have one copy (even for data) regardless of how many objects are created (same idea used in C)

Initialization

When the constructor is called, `this` pointer points to an **uninitialized** block of memory

It's the job of the constructor to initialize this memory properly

Constructor has the same name as the class, and can be overloaded with a different parameter list - the proper one will be called automatically

`this` Pointer

Why do we need `this`?

When local variable name overlaps with member name

Return reference to the calling object

- Chained function calls!

this Pointer

```
class Test
{
private:
  int x; int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test& setX(int a) { x = a; return *this; }
  Test& setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
int main()
{
  Test obj1(5, 5);
  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

Cleanup

In C, only thing you need to worry about in terms of cleanup is heap memory (created through `malloc()`)

Other data types (`int`, `float`, etc.) are usually ignored (automatically cleaned up, like in C)

However, in C++, you use more complex data types (i.e., classes), and so proper cleanup is often required

- Destructor is called (automatically) when the object goes out of scope (e.g., return from a function)
  - Destructors are named with ~<class name>, and have no arguments (you don't need options to destroy things)

Example

```
class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};
Tree::Tree(int initialHeight) {
  height = initialHeight;
}
Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}
void Tree::grow(int years) {
  height += years;
}
void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}

int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
} ///:~
```

What is the output?

Example

```
before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace
```

Storage Allocation

In C++, when an object is created, it is simultaneously initialized (with a constructor)

- This ensures you have no uninitialized objects in your code
- You will NOT be allowed to create an object before you have the initialization Information for the constructor

However, the compiler will allocate storage for an object at the **beginning of its scope** (but you can't access this until it has been **defined**, e.g., `Human John;`)

Constructor call will NOT happen until the code reaches that definition statement

- As such, the compiler will NOT let you put the object definition where it may not pass through (e.g., because of a conditional statement)

Example

```
1.  class X {
2.  public:
3.    X();
4.  };
5.  X::X() { cout << "X constructor" << endl;}
6.
7.  void f(int i) {
8.    if(i < 10) {
9.        goto jump1;
10.   }
11.   X x1;
12.   jump1:
13.   switch(i) {
14.     case 1 :
15.       X x2;
16.       break;
17.     case 2 :
18.       X x3;
19.       break;
20.   }
21. }
```

Example

```
1.  class X {
2.  public:
3.    X();
4.  };
5.  X::X() { cout << "X constructor" << endl;}
6.
7.  void f(int i) {  // x1 storage allocated here
8.    if(i < 10) {
9.        goto jump1;  // skips x1 constructor - error
10.   }
11.   X x1;
12.   jump1:
13.   switch(i) {  // x2 and x3 storage allocated here
14.     case 1 : // case bypasses x3 constructor
15.       X x2;  // x2 constructor called here
16.       break;
17.     case 2 : // case bypasses x2 constructor
18.       X x3;  // x3 constructor called here
19.       break;
20.   }
21. }
```

Constructors and Inheritance

Order of constructor call

- Base class constructors are always called first in the derived class' constructor
- Then, the derived class' constructor is called

This is because the derived class' constructor has access to only its own members

- The derived class may have also inherited properties of the base class, and only the base class' constructor can initialize them

Constructors and Inheritance

```
class Base
{
    int x;
    public:
    // default constructor
    Base() { cout << "Base default constructor\n"; }
};
class Derived : public Base
{
    int y;
    public:
    // default constructor
    Derived() { cout << "Derived default constructor\n"; }
    // parameterized constructor
    Derived(int i) { cout << "Derived parameterized constructor\n"; }
};

int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```

Constructors and Inheritance

```
Base default constructor
Base default constructor
Derived default constructor
Base default constructor
Derived parameterized constructor
```

Constructors and Inheritance

You can also invoke the base class' constructor

```
Base::Base(int i) {
    x = i;
    cout << "Base constructor\n";
}
```

```
class Derived : public Base
{
    int y;
    public:
    // default constructor
    Derived() { cout << "Derived default constructor\n"; }
    // parameterized constructor
    Derived(int i) : Base(i) // Base::x is private, but will be
initialized to with i by using its own constructor
    {
        y = j;
        cout << "Derived parameterized constructor\n";
    }
};
```

Constructors and Inheritance

This allows some members to remain private in the Base class

- But can still be initialized and accessed (if the proper interface has been defined)
- Code remains modularized (e.g., you won't have to initialize some protected variable in both the base and derived class constructors)